**Bounce Quickstart Guide**

Copyright © Irlan Robson

Content

[1. Introduction 3](#_Toc523755469)

[1.1. Help 3](#_Toc523755470)

[1.2. Basic Concepts 3](#_Toc523755471)

[1.3. System of Units 4](#_Toc523755472)

[2. Hello World! 5](#_Toc523755473)

[2.1. Including Bounce headers 7](#_Toc523755474)

[2.2. Setting profile callbacks 8](#_Toc523755475)

[2.3. Creating the world 8](#_Toc523755476)

[2.4. Creating the ground body 9](#_Toc523755477)

[2.5. Creating the ground shape 9](#_Toc523755478)

[2.6. Creating the dynamic body 11](#_Toc523755479)

[2.7. Stepping the world 11](#_Toc523755480)

[2.8. Visualizing the simulation results 12](#_Toc523755481)

[2.9. Destroying the world 12](#_Toc523755482)

[3. End 13](#_Toc523755483)

# Introduction

Bounce is a C++ 3D physics engine for games. The library can be used for convex hull generation, collision detection, and physics simulation of cloth and rigid bodies.

For using Bounce, you’ll need

* be experienced with C++ programming,
* a good background in 3D mathematics, and
* familiarity with essential physics concepts such as time, position, velocity, force, and impulse.

## Help

The present Quickstart Guide (QSG) is not a complete user manual. If you need further help or want to collaborate with the project you can open an issue in the GitHub issue tracker at <https://github.com/irlanrobson/bounce>.

Installation instructions are available in the repository above as well.

## Basic Concepts

**Convex Hull**

A convex hull is a geometrical object. For any two given points on a convex hull the line between these two points is also contained inside the convex hull. Examples of convex hulls are spheres, capsules, and boxes.

**Collision Shape**

Thus, a geometrical object which contains only geometrical and topological information of that object. A convex hull, for example, thus is a collision shape.

**Shape**

A physics shape is a geometrical object. It extends a collision shape so it also contains physical properties such as density, friction, and restitution.

**Rigid Body**

A rigid body is a solid body. The distance between any two given points on a rigid body is constant over time independently of how much force is exerted on it. Think of a rock. Each 3D rigid body has 6 degrees of freedom. 3 linear degrees of freedom and 3 angular degrees of freedom.

**Constraint**

A constraint removes the relative degrees of freedom between two or more rigid bodies.

**Cloth**

Deformable surface usually represented by a collection of particles connected with each other via springs or constraints. Cloth is typically defined from a triangle mesh.

**Contact**

A contact is a constraint between two shapes that exists when the shapes are overlapping with each other. A sphere slipping over a plane is constrained to move only tangentially to that plane.

**Joint**

Constraint between two or more rigid bodies. A door attached to a wall by a hinge is constrained to rotate about the axis of rotation of the hinge.

**World**

A world is a set of physics objects such as cloth, rigid bodies, and constraints. Each object can be affected by the world gravitational force.

## System of Units

Bounce uses the MKS system of units. The MKS system has metre, kilogram, and seconds as its base units.

Note: Angles have units of radians, not degrees.

# Hello World!

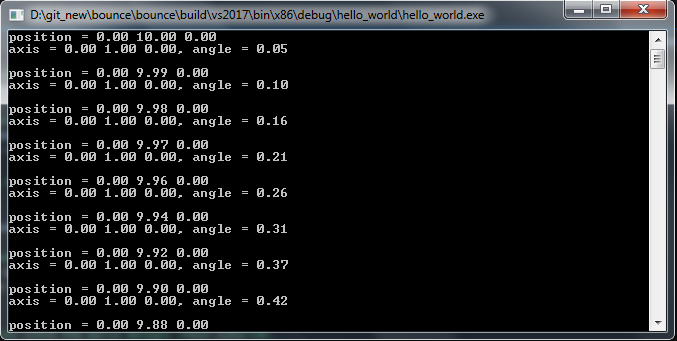
****

Figure 1: Hello world console program output.

Probably the easiest route to take in order to learn how to use a library is creating a program that uses some of its basic features. Let us begin by writing a very simple program in C++ which runs 1 second of simulation. For simplicity, we will be using printf for rendering the simulation results. This program is available in the examples folder inside the project repository.

|  |
| --- |
| #include <bounce/bounce.h>  #include <stdio.h>  // We don't care for a profiler. This definition does nothing.  void b3BeginProfileScope(const char\* name)  {  }  // We don't care for a profiler. This definition does nothing.  void b3EndProfileScope()  {    }  // This example shows how to setup and run a simple simulation  // using Bounce.  int main(int argc, char\*\* argv)  {  // The world. We allocate it using the heap but you can to it  // on the stack if the stack is sufficiently large.  b3World\* world = new b3World();  // The world gravity.  const b3Vec3 gravity(0.0f, -9.8f, 0.0f);  world->SetGravity(gravity);    // The fixed time step size.  const float32 timeStep = 1.0f / 60.0f;    // Number of iterations for the velocity constraint solver.  const u32 velocityIterations = 8;  // Number of iterations for the position constraint solver.  const u32 positionIterations = 2;  // Create a static ground body at the world origin.  b3BodyDef groundDef;  b3Body\* ground = world->CreateBody(groundDef);  // Create a box positioned at the world origin and  // aligned with the world frame.  b3BoxHull groundBox;    // Set the ground box dimensions using a linear scale transform.  b3Transform scale;  scale.position.SetZero();  scale.rotation = b3Diagonal(10.0f, 1.0f, 10.0f);  groundBox.SetTransform(scale);    // Create the box physics wrapper.  b3HullShape groundShape;  groundShape.m\_hull = &groundBox;  // Add the box to the ground body.  b3ShapeDef groundBoxDef;  groundBoxDef.shape = &groundShape;  ground->CreateShape(groundBoxDef);  // Create a dynamic body.  b3BodyDef bodyDef;  bodyDef.type = e\_dynamicBody;    // Position the body 10 meters high from the world origin.  bodyDef.position.Set(0.0f, 10.0f, 0.0f);    // Set the initial angular velocity to pi radians (180 degrees) per second.  bodyDef.angularVelocity.Set(0.0f, B3\_PI, 0.0f);    b3Body\* body = world->CreateBody(bodyDef);  // Create a unit box positioned at the world origin and  // aligned with the world frame.  b3BoxHull bodyBox;  bodyBox.SetIdentity();  // Create the box physics wrapper.  b3HullShape bodyShape;  bodyShape.m\_hull = &bodyBox;  // Add the box to the body.  b3ShapeDef bodyBoxDef;  bodyBoxDef.shape = &bodyShape;  bodyBoxDef.density = 1.0f;  body->CreateShape(bodyBoxDef);  // Run a small game loop of 60 frames length.  for (u32 i = 0; i < 60; ++i)  {  // Perform a time step of the world in this frame.  world->Step(timeStep, velocityIterations, positionIterations);    // Read the body position and orientation in this frame.  b3Vec3 position = body->GetPosition();  b3Quat orientation = body->GetOrientation();    // Decode the axis and angle of rotation about it from the quaternion.  b3Vec3 axis;  float32 angle;  orientation.GetAxisAngle(&axis, &angle);  // Visualize the body state in this frame.  printf("position = %.2f %.2f %.2f\n", position.x, position.y, position.z);  printf("axis = %.2f %.2f %.2f, angle = %.2f\n\n", axis.x, axis.y, axis.z, angle);  }    // Now destroy the bodies since the world manages their lifetime.  delete world;  return 0;  } |

Now we are going to explain step-by-step each part of the code.

## Including Bounce headers

We first start by including the main project header which is bounce.h. We also include stdio.h in order to use printf later.

|  |
| --- |
| #include <bounce/bounce.h>  #include <stdio.h> |

## Setting profile callbacks

Bounce does not profile the code internally. Rather it just marks which section of code should be profiled and lets the user to handle this task further. When a scope that is marked to be profiled is reached, it calls the following user-defined function.

|  |
| --- |
| void b3BeginProfileScope(const char\* name); |

The parameter name is the profile scope identifier.

Correspondingly, the function below is called at the end of the scope.

|  |
| --- |
| void b3EndProfileScope(); |

These two functions allow the user to measure the elapsed time of the profiled section and store them for visualization, for instance.

Because in this example project we don’t use a profiler, simply define those functions both that just do the return.

|  |
| --- |
| void b3BeginProfileScope(const char\* name)  {  }  void b3EndProfileScope()  {    } |

## Creating the world

In Bounce a world is an object managing all the physics objects and also exposing interfaces for the user to perform different types of physics queries. Every program that uses Bounce requires the user to create (and destroy) a world.

|  |
| --- |
| b3World\* world = new b3World(); |

You set the world acceleration of gravity. The function used to set the world acceleration of gravity is

|  |
| --- |
| b3World::SetGravity(const b3Vec3& gravity); |

For example, near the surface of the Earth each object accelerates towards the Earth center of mass by approximately 9.8 m/s^2. Units are meters per second squared. Here we set gravity to the vector (0, -9.8, 0) assuming that the world surface is the xy plane.

|  |
| --- |
| const b3Vec3 gravity(0.0f, -9.8f, 0.0f);  world->SetGravity(gravity); |

Next we need define the simulation parameters. First we must set a time-step for the simulation. A time-step is how many seconds the simulation should be advanced at each step. For performing a single time-step the following function needs to be called.

|  |
| --- |
| void b3World::Step(float32 timeStep, u32 velocityIterations, u32 positionIterations); |

It is recommended to use a small and constant time step during the simulation time for several reasons. Mainly because of simulation realism, determinism, and stability. Remember that thus time has units of seconds.

|  |
| --- |
| const float32 timeStep = 1.0f / 60.0f; |

In order to satisfy constraints (due to joints and contacts) while reaching good performance, Bounce uses iterative constraint solvers. The iterative constraint solvers require defining the number of solver iterations. The velocity solver corrects violated velocities of rigid bodies while the position solver fixes violated positions of rigid bodies. Basically the number of iterations depends on the simulation configuration. For example, a tall stack of boxes will probably require a large number of velocity iterations in order to remain stable. A small number of iterations can make the stack fall. This is not realistic.

|  |
| --- |
| const u32 velocityIterations = 8;  const u32 positionIterations = 2; |

Now that we have set up the world let us add some rigid bodies to it.

## Creating the ground body

Bodies are created and destroyed by the following functions.

|  |
| --- |
| b3Body\* b3World:: CreateBody(const b3BodyDef& def);  void b3World::DestroyBody(b3Body\* body); |

Body creation requires you to define the initial configuration of the body using a body definition. b3BodyDef is the body definition. By default, the body definition is configured to set the body to a static body and located at the origin (0, 0, 0).

|  |
| --- |
| b3BodyDef groundDef;  b3Body\* ground = world->CreateBody(groundDef); |

## Creating the ground shape

In the example code we create a simple box shape for the ground body using the object b3BoxHull.

b3BoxHull is a collision shape that extends a base b3Hull object (a convex hull). One way for the user to manipulate the box dimensions is by setting the box from an affine transformation. Set the box from a transformation by calling the following function giving the transformation as parameter.

|  |
| --- |
| void b3BoxHull::SetTransform(const b3Transform& transform); |

If you don’t need to rotate the box then you can set from extension using the function

|  |
| --- |
| void b3BoxHull::Set(float32 ex, float32 ey, float32 ez); |

The code above requires the user to pass the box extensions along the standard axes as parameters.

The following code creates a box with dimensions 10 x 1 x 10 m, located at the origin, and aligned with the world frame.

|  |
| --- |
| b3BoxHull groundBox;    b3Transform scale;  scale.position.SetZero();  scale.rotation = b3Diagonal(10.0f, 1.0f, 10.0f);  groundBox.SetTransform(scale); |

The box that it was just created doesn’t contain physics properties such as density, friction, and restitution. It just contains geometrical information for its associated physics shape that we will need to create in order to let the rigid body create the shape.

|  |
| --- |
| b3HullShape groundShape;  groundShape.m\_hull = &groundBox; |

Shapes are created and destroyed using the following functions.

|  |
| --- |
| b3Shape\* b3Body::CreateShape(const b3ShapeDef& def);  void b3Body::DestroyShape(b3Shape\* shape); |

The first function needs a shape definition to be passed as parameter. In the example we have passed our box resource to a temporary physics shape that is used only to define the rigid body using the shape definition.

|  |
| --- |
| b3ShapeDef groundBoxDef;  groundBoxDef.shape = &groundShape;  ground->CreateShape(groundBoxDef); |

The function b3Body::CreateShape returns a pointer to an abstract b3Shape object. In this example code, however, the created shape is not used externally so we don’t need to keep track of the shape pointer.

Note: Shapes are positioned relative to the rigid body. You can add multiple shapes per body. You can also reuse collision shapes. For example, you can create another shape for the rigid body using the same box hull that we’ve defined previously.

## Creating the dynamic body

Similarly to the previous example, we can create a dynamic rigid body and attach a shape to it. However, before creating the body we must set b3BodyDef::type to e\_dynamicBody.

We also must configure the shape definition to have a positive density. Simply set b3ShapeDef::density to a positive value. Remember that density has units of kilograms per cubic meter (kg/m^3).

|  |
| --- |
| b3BodyDef bodyDef;  bodyDef.type = e\_dynamicBody;    bodyDef.position.Set(0.0f, 10.0f, 0.0f);  bodyDef.angularVelocity.Set(0.0f, B3\_PI, 0.0f);    b3Body\* body = world->CreateBody(bodyDef);  b3BoxHull bodyBox;  bodyBox.SetIdentity();  b3HullShape bodyShape;  bodyShape.m\_hull = &bodyBox;  b3ShapeDef bodyBoxDef;  bodyBoxDef.shape = &bodyShape;  bodyBoxDef.density = 1.0f;    body->CreateShape(bodyBoxDef); |

## Stepping the world

Now that world is configured you can perform a simulation step by calling the function below. Calling this function advances the simulation by the amount of time-step given.

|  |
| --- |
| world->Step(timeStep, velocityIterations, positionIterations); |

## Visualizing the simulation results

Having executed b3World::Step the dynamic rigid body we created in the last section has been moved downwards due to acceleration of gravity. Now it’s a good time to see the simulation results in the screen.

|  |
| --- |
| b3Vec3 position = body->GetPosition();  b3Quat orientation = body->GetOrientation();    b3Vec3 axis;  float32 angle;  orientation.GetAxisAngle(&axis, &angle);    printf("position = %.2f %.2f %.2f\n", position.x, position.y, position.z);  printf("axis = %.2f %.2f %.2f, angle = %.2f\n\n", axis.x, axis.y, axis.z, angle); |

This part of code retrieves the position and orientation of the rigid body at the current frame. printf outputs the results in the console.

Note: Bounce uses quaternions and 3x3 matrices interchangeably for representing rotations. While it does not requires the user to have deep knowledge of quaternions, basic notion of it might help when programming with the library. The library has convenience functions for converting between those representations, so no need to worry in practice if you’re familiar only with rotation matrices.

## Destroying the world

Destroying the world will destroy all the physics objects that were created in the previous sections, such as rigid bodies and shapes.

|  |
| --- |
| delete world; |

# End

Congratulations! You’ve reached the end of Bounce Quickstart Guide. For further information about the project please visit <https://github.com/irlanrobson/bounce>.